1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery`:

```
algoritmo mystery(n: Int) --> Int
  k=1
  while (k<n)
    k=k+2
  endwhile
  if (k==1)
    return 1000
  else
    return mystery2(n/2)+mystery(n/4)
  endif

algoritmo mystery2(k:Int) --> Int
  return mystery(k/2)
```

Soluzione Nell'algoritmo `mystery` tutte le operazioni richiedono tempo costante, meno le due chiamate; una ricorsiva indiretta tramite la funzione ausiliaria `mystery2`, ed una ricorsiva diretta. Il caso base della ricorsione si ha quando il parametro `n` è minore o uguale a 1. Il cosctrutto `while` esegue $n/2 = O(n)$ cicli. Le due chiamate ricorsive sono bilanciate, utilizzando $n/4$ come valore del parametro: infatti, la chiamata indiretta prima considera $n/2$ quando chiama `mystery2`, che a sua volta invoca `mystery` dividendo per due il parametro. Si ottiene quindi la seguente equazione di ricorrenza

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2T(n/4) + n & \text{altrimenti} \end{cases}$$

Applicando il master theorem otteniamo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 4} = \frac{1}{2}$ e $\beta = 1$. Avendo $\alpha < \beta$, applichiamo il terzo caso del master theorem ottenendo $T(n) = O(n^\beta) = O(n)$.

2. Dato un *albero AVL* inizialmente vuoto, effettuare le seguenti operazioni in ordine e mostrare lo stato dell'albero dopo ogni operazione: INSERT(30); INSERT(50); INSERT(60); INSERT(65); DELETE(50); DELETE(30); INSERT(10); INSERT(70); INSERT(62); INSERT(63).

Soluzione Vedi ultima pagina.

3. Progettare un algoritmo che, dato un vettore di n numeri ordinati in senso non decrescente, un numero x, ed un numero $k \geq 0$, indica quanti numeri del vettore differiscono da x per meno di k (in altri termini, quanti numeri y del vettore sono tali che $|y - x| \leq k$).

Soluzione Il problema equivale a cercare il numero di elementi compresi fra $x - k$ e $x + k$. È possibile adottare una soluzione divide-et-impera che divide il sottovettore da controllare solo se l'intervallo tra il valore minimo e massimo del sottovettore contiene uno dei due estremi dell'intervallo $[x-k, x+k]$. In caso contrario, infatti, si sa che o tutti gli elementi sono nell'intervallo, oppure nessuno lo è. Quindi non è necessario dividere ulteriormente il problema in quanto si conosce già la soluzione.

Risolviamo il problema tramite il seguente algoritmo ricorsivo, che oltre ai tre parametri delle specifiche, contiene i parametri s ed e che indicano inizio e fine del sottovettore da considerare. Inizialmente invocheremo l'algoritmo con `conta(v,x,k,1,n)`.

```
algoritmo conta(number v[1..n], number x, number k, int s, int e) --> int
  if (s>e) or (v[s] > x+k) or (v[e] < x-k)
    return 0
  elseif (v[s] >= x-k) and (v[e] <= x+k)
    return e-s+1
  else
    int m = (s+e)/2
    return conta(v,x,k,s,m)+conta(v,x,k,m+1,e)
```

Procediamo ora all'analisi del costo computazionale. Ad ogni chiamata ricorsiva, la quantità di elementi nel sottovettore da considerare viene dimezzata. Quindi, al massimo, avremo $O(\log n)$ chiamate ricorsive annidate. Abbiamo però che ad ogni livello di annidamento, al più due chiamate effettueranno le chiamate al livello di annidamento successivo. Tali chiamate sono quelle per cui l'intervallo tra il valore minimo e massimo del sottovettore contiene uno dei due estremi dell'intervallo $[x-k, x+k]$. Complessivamente il numero di chiamate sarà quindi $O(\log n)$. Considerando che le operazioni all'interno della funzione, ad esclusione delle chiamate ricorsive, risulta di costo costante, possiamo concludere che $T(n) = O(\log n)$.

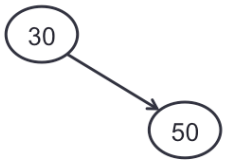
4. Si consideri un impianto di irrigazione composto da tubature e giunti posti ai punti terminali delle tubature. Ogni tubatura ha un proprio tempo di attraversamento, ovvero il tempo che l'acqua impiega per attraversarlo. L'impianto è rappresentato tramite un grafo non orientato pesato $G = (V, E, w)$ dove V è l'insieme dei giunti, E è l'insieme delle tubature (la tubatura che collega il giunto u al giunto v è rappresentata dalla coppia –non ordinata– $\{u, v\}$), e w è la funzione di costo che, data una tubatura adiacente ai giunti $u, v \in V$, $w(\{u, v\})$ indica il tempo necessario per attraversare tale tubatura. Anche i giunti richiedono un certo tempo T per essere attraversati. Uno dei giunti $s \in V$ è collegato ad un rubinetto da cui viene erogata l'acqua che deve scorrere nell'impianto. Progettare un algoritmo che dato il grafo $G = (V, E, w)$ che rappresenta l'impianto, il tempo di attraversamento dei giunti T, e il giunto collegato al rubinetto s, restituisce il tempo che intercorre dall'apertura del rubinetto al riempimento completo dell'impianto.

Soluzione Il problema può essere risulto utilizzando una versione modificata dell'algoritmo di Dijkstra in cui si tiene conto anche del costo relativo all'attraversamento dei giunti. Al termine dell'esecuzione dell'algoritmo, si restituisce la distanza massima di un giunto, ovvero il tempo necessario per l'acque per raggiungere anche l'ultimo giunto.

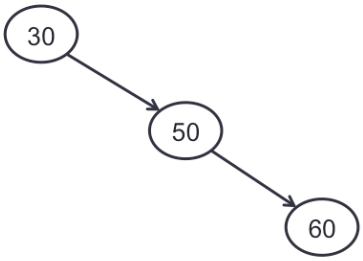
```
algoritmo RiempiImpianto (Grafo G=(V,E,w), double T, int s) --> double

  // inizializzazione strutture dati
  int n = G.numNodi()
  double D[1..n]
  for v = 1..n do
    D[v] = INFINITY
  endfor
  D[s] = T
  CodaPriorita<int, double> Q; Q.insert(s, D[s]);

  // esecuzione algoritmi di Dijkstra
  while (not Q.isEmpty()) do
    u = Q.find(); Q.deleteMin()
    D[u] = D[u] + T
    for each v adiacente a u do
      if (D[v] == INIFINITY) then
        // prima volta che si incontra v
        D[v] = D[u] + w(u,v)
        Q.insert(v, D[v]);
      elseif (D[u] + w(u,v) < D[v]) then
        // scoperta di un cammino migliore per raggiungere v
        Q.decreaseKey(v, D[v] - D[u] - w(u,v))
        D[v] = D[u] + w(u,v)
      endif
    endfor
  endwhile
  return D[u]
```

La complessità dell'algoritmo è la medesima dell'algoritmo di Dijkstra, ovvero $T(n, m) = O(m \log n)$, dove n è il numero di vertici ed m il numero di archi nel grafo.

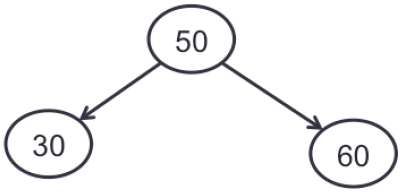
Cont.



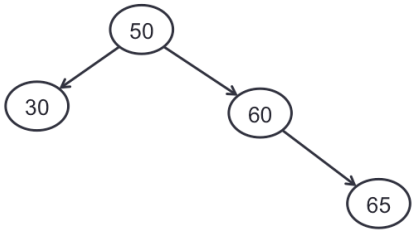
(a) INS(30) e INS(50). Inizialmente 30 è la radice. 50 diventa figlio destro di 30 per mantenere le proprietà di un ABR.



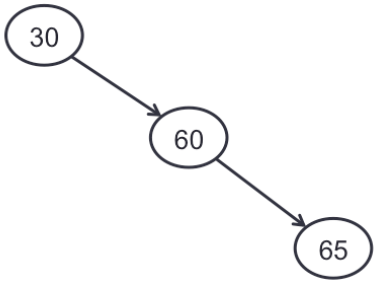
(b) INS(60). 60 diventa figlio destro di 50 per mantenere le proprietà di un ABR. L'albero risulta sbilanciato: 30 ha fattore di bilanciamento -2.



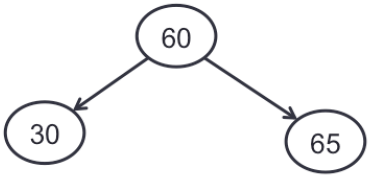
(c) Rotazione sinistra per bilanciare l'albero AVL.



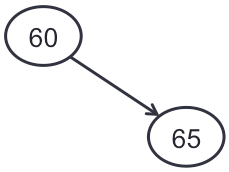
(d) INS(65). 65 diventa figlio destro di 60 per mantenere le proprietà di un ABR.



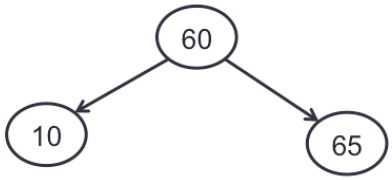
(e) DEL(50). Viene eliminata la radice e il suo predecessore 30 diventa la nuova radice. L'albero risulta sbilanciato: 30 ha fattore di bilanciamento -2.



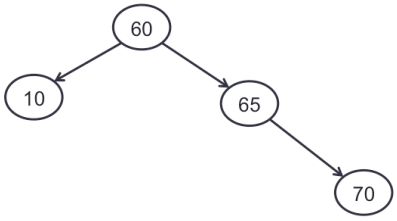
(f) Rotazione sinistra per bilanciare l'albero AVL.



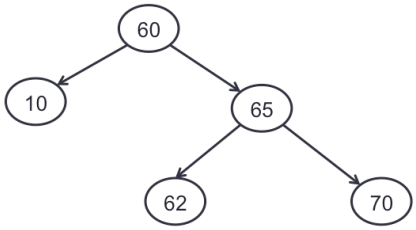
(g) DEL(30). L'elemento 30 viene eliminato e non sono richieste altre operazioni.



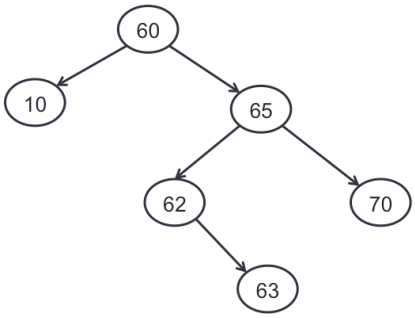
(h) INS(10). 10 diventa figlio sinistro di 60 per mantenere le proprietà di un ABR.



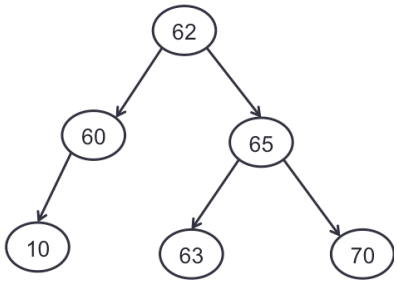
(i) INS(70). 70 diventa figlio destro di 65 per mantenere le proprietà di un ABR.



(j) INS(62). 62 diventa figlio sinistro di 65 per mantenere le proprietà di un ABR.



(k) INS(63). 63 diventa figlio destro di 62 per mantenere le proprietà di un ABR. L'albero risulta sbilanciato: 60 ha fattore di bilanciamento -2.



(l) Rotazione doppia (destra-sinistra) per bilanciare l'albero AVL.

The End.